2020

ANALYZING THE PERFORMANCE IMPACT OF PARALLEL LATENCY IN THE PIPELINE

Matthew Constant
University of Rhode Island, mconstant3496@gmail.com

ANALYZING THE PERFORMANCE IMPACT OF PARALLEL LATENCY IN THE PIPELINE

BY

MATTHEW CONSTANT

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2020

MASTER OF SCIENCE THESIS

OF

MATTHEW CONSTANT

APPROVED:

Thesis Committee:

Major Professor Resit Sendag

Bin Li

Lutz Hamel

Nasser Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2020

ABSTRACT

This work introduces a concept coined *overlap latency* which is shown to severely limit performance in several types of benchmarks. This overlap latency is only completely removed when both branch mispredictions and cache misses are removed in tandem, rather than improved in isolation. Since most current research investigates improvements to branch predictions or cache behavior - and not both - proposed techniques are not able to unlock this extra performance gain. To demonstrate this concept benchmarks are evaluated using four configurations: baseline which uses current state-of-the-art branch prediction and cache prefetching, perfect-bp which emulates perfect branch prediction direction, perfect-cache which emulates a perfect L1 data cache, and perfect which combines perfect-bp and perfect-cache. In addition, detailed analysis on select benchmarks is conducted to show the cause of overlap latency as well as the effect this has on an out-of-order execution CPU. Benchmarks were found to have the potential for up to an additional 229% IPC compared to that expected based on individual performance gain from branch prediction and cache.

ACKNOWLEDGMENTS

I would like to acknowledge all those who have helped me to achieve this academic accomplishment. Specifically, I would like to thank Dr. Sendag for his advice and guidance as well as the opportunities he has provided me. I would like to thank Tyler Tucker for his help in conducting experiments and gathering results. Thank you to all my committee members for all of the time and advice given to me.

I would also like to thank my family and friends for their unwavering support. Finally, I would like to thank all of my fellow students who have helped, motivated, and inspired me along me the way.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

CHAPTER 1

Introduction

Since the invention of the multi-stage pipeline[1], through the widely adopted Out-of-Order (O3) execution model[1], and continuing today with active research in branch prediction[2] and data cache prefetching[3] - among many other areas - computer architecture has consistently focused on improving single core performance. Even with promising contributions from newer research topics (such as multi-core architectures[4] and hardware accelerators[5]) which provide opportunities for new and creative innovations in computer architecture, single core performance remains among the most important sources of improvement[6]. This can be seen in industry, where microprocessor vendors have been devoting significant hardware resources - including deeper pipelines and wider issue and commit widths - to push the limits of single core performance. This trend is also seen in academia where competitions are regularly organized to find and refine branch prediction[7] and cache prefetching[8] techniques. Even outside of competitions, many papers are proposed every year in this area of research.

This high level of interest and activity has led to great improvements to single core performance. However, continuing to produce more and more effective branch prediction and data cache prefetching is becoming much more difficult. For example, two main sources of branch mispredictions and cache misses in today's state-of-the-art processors can be attributed to hard to predict (H2P) branches[9] and irregular data accesses[10, 11], respectively. Since these do not follow some repetitive pattern, which current mechanisms utilize to predict future behavior, new and innovation solutions will be needed to predict them with any degree of accuracy and coverage. In addition to this, H2P branches and irregular data accesses can of-

ten be interrelated.For example, a pointer-based data structure traversal (irregular data accesses) cannot be effeciently prefetched without accurate branch prediction. Conversely, branches which depend on irregular data accesses take much longer to resolve on a cache miss which can result in more squashed instructions.

In this work, the co-dependence between H2P branches and irregular data accesses is evaluated in a variety of benchmark suites to determine the degree to which this behavior effects single core performance. In addition, a detailed analysis is carried out to determine the causes (software implementations) and effects (architectural strains) of the load-branch relationship. This study makes the following contributions:

- Introduce the concept of overlap latency, a consequence of load-branch dependencies in frequently executed loops.
- Provide an upper bound performance limit on a variety of benchmarks to show the limitations in improving branch prediction and cache performance in isolation due to overlap latency.
- Categorize selected benchmarks into groups of algorithms which are vulnerable to overlap latency.
- Investigate the cause of overlap latency and its effects on a modern processor.

List of References

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[2] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, 2019.

[3] S. Mittal, "A survey of techniques for dynamic branch prediction," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, p. e4666, 2019.

[4] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: preparing for a new exponential," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 67–72.

[5] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 151–160.

[6] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[7] "Championship branch prediction (cbp-5)," 2016. [Online]. Available: https://www.jilp.org/cbp2016/

[8] "The 3rd data prefetching championship," 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/

[9] C.-K. Lin and S. J. Tarsa, "Branch prediction is not a solved problem: Measurements, opportunities, and future directions," *arXiv preprint arXiv:1906.08170*, 2019.

[10] P. Braun and H. Litz, "Understanding memory access patterns for prefetching," in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, 2019.

[11] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 513–526.

CHAPTER 2

Literature Review

2.1 Background

This section will provide a background on important topics related to this research, as well as a survey of related works.

2.1.1 Pipelining

A pipelined computer architecture takes advantage of parallelism inherent in a set of instructions by overlapping the execution of the instructions. A pipeline essentially breaks the processing of an instruction into several stages. Each of these stages take less time to complete then the entire processing stage would, allowing for increased clock cycles. In addition, it allows for several instructions to be processed simultaneously since the instructions can be in different pipeline stages.

While this technique has led to substantial performance gains, it also creates a more complex environment to process instructions in. For example, if an instruction depends on the results of the previous instructions, the instruction must wait to begin execution until the result of the previous instruction is available. In the event the previous instruction's result depends on a memory access, this wait could be many CPU cycles long. This waiting is referred to as a pipeline stall, and results in underutilization of the CPU resources.

Another complication introduced by the pipeline architecture is referred to as a control hazard. Control hazards are the result of branch instructions in which the next instruction to execute depends on the result of the current instruction. The most simple solution to a control hazard is to stall the pipeline until it is known what the next instruction should be. Since branches represent about one-third of

all instructions, this stalling can result in significant performance loss.

2.1.2 Out-of-Order Execution

Out-of-Order (O3) execution is used in almost all modern architectures. This allows for instructions to be executed out of program order however still committing instructions in the correct order. This is useful for several reasons, but mainly to hide memory accesses. As mentioned in the previous section, if an instruction depends on the previous instruction's result it must wait for the result. However, there may be other instructions independent of these two instructions which could occupy the pipeline instead. O3 execution allows for instructions to be executed as soon as they are ready to execute and then reorders the executed instructions back into program order to ensure correct program execution. This reordering is accomplished through the use of a reorder buffer (ROB) which stores all executed instructions until they are able to be committed in program order.

2.1.3 Data Prefetching

While O3 execution is effective at hiding memory latency by allowing the execution of other instructions while waiting, data prefetching is a technique used to actually remove these memory latencies. This is done by attempting to predict near-future memory accesses, and storing the data in the cache before the program requires the data. When predicted correctly, prefetching removes what would otherwise result in a cache miss. There are many very effective methods of data cache prefetching including: Stride Prefetching[1], Signature Path Prefetching[2], Best Offset Prefetching[3] and Indirect Memory Prefetching[4].

Irregular Data Accesses

While data cache prefetching has provided significant improvement to cache performance, and therefore program performance, there remain memory accesses

which even the state-of-the-art designs cannot predict. For example, an irregular memory access can occur when traversing a linked data structure in which the values fetched are pointers to random memory locations on the heap. Since almost all current data prefetchers utilize some history or repetitive pattern to make predictions, this type of traversal is very difficult to be predicted. There have been several proposed designs to handle the irregular memory access problem[5, 6, 7].

2.1.4 Branch Prediction

As mentioned in the pipeline section, stalls due to control hazards can significantly limit a program's performance. Branch prediction (BP) is a technique used in all modern CPUs which attempts to predict whether a branch will be taken or not rather than wait for the actual outcome. The CPU then proceeds to execute instructions assuming the prediction is correct. If the prediction turns out to be wrong, the state of the pipeline must be reverted back to the mispredicted instruction and execution resume down the correct path. Many effective branch prediction techniques have been proposed, however most CPUs use either the TAGE[8] predictor or Perceptron-based prediction[9]. While these branch predictors are able to acheive over 90% accuracy many cases, there remain branches which cannot be accurately predicted. This problem has been investigated, and is referred to as hard to predict (H2P) branches.

2.1.5 Microarchitecture Simulation

In order to test new microarchutectural ideas and implementations, it is common to use software to emulate the functionality of the proposed hardware. This allows for changes to the architecture to be made quickly as well as for the ability to record fine grain details about the implementation. However, software emulating hardware runs orders of magnitude slower than the actual hardware[10]. With

today's ever increasing complexity in both the microarchitecture and the software that is run on it, simulations times can take days, weeks, or even months.

Functional vs Detailed Simulations

The effort to reduce the required simulation time while maintaining accurate and detailed results continues to be an active research area. A fundamental trade-off is between functional and detailed simulations. A functional simulation will emulate the essential functionality of the hardware while optimizing or removing other parts of the simulation in order to reduce simulation time. A detailed simulation will accurately emulate the hardware at a very low level, providing useful and insightful statistics at the cost of much longer simulation times.

A common technique used is to carry out a functional simulation of some workload on the proposed hardware and take a checkpoint at some region of interest. This checkpoint will store essential information about architectural state and memory contents so that future simulations can begin at this point. This checkpoint is then restored using a detailed simulation which will simulate some predetermined number of instructions. This provides a compromise, assuming there is a region of interest that is known before simulation. In cases where this assumption does not hold true, other checkpointing methodologies have been proposed.

Checkpointing Methods

The two most commonly used checkpointing methodologies are Simpoint[11] and SMARTS[12]. Simpoint requires a functional simulation to be conducted from start to finish in which a basic block vector analysis is used. The workload is then broken into sections and weights are assigned to each section. Detailed simulations can then be carried out on each section, with the overall results being merged according to each section's assigned weight. Simpoint is effective because it is able

to remove redundent or similarly behaving areas of the workload, thus reducing the number of instructions which must be simulated in detail. A disadvantage of Simpoint is that once the sections (assigned by instruction number) are determined (first functional simulation pass), a second full functional simulation is required to take checkpoints at the specific instructions.

The SMARTS methodology breaks the workload into some number of even intervals with each section being assigned an equal weight. In this case, only one functional simulation needs to be run to take checkpoints at each interval. A disadvantage of this method is that it can take many trials to find a sufficient interval length such that the averaged results approach the actual results. In addition, SMARTS typically requires a large number of checkpoints to be taken which can lead to significant storage space.

2.2 Related Works

While a majority of active research in branch prediction and data prefetching rely on some kind of history to predict the future accesses, there have been novel approaches which attempt to resolve H2P branches and irregular data accesses.

Control Flow Decoupling[13] (CFD) proposes separating loops into two loops, one containing all predicate instructions necessary to determine the branch outcome and another containing all control-dependent instructions. While this transformation is only possible on a subset of loops which contain certain characteristics, it is able to totally remove branch mispredictions for these loops. This is done by recording the outcome of each set of predicate instructions and storing the branch outcome into a queue which is used by the CPU to determine if the control dependent instructions should be executed for each iteration. This is able to remove H2P branch mispredictions when these branches are present in loops which are suitable for such transformation.

Slipstream[14] is a novel architecture proposal in which two versions of the same program are executed simultaneously. In one version, unimportant (to program outcome) instructions are identified at runtime and skipped, while the other version executes all instructions. The two instruction streams are able to communicate to each other, allowing the shorter version to convery necessary information - like branch outcomes and memory information - to the full version. This is possible since the shorter version executes ahead of the full version due to skipping some instructions.

Runahead execution[15] is another method of executing instructions ahead of the actual instruction stream. In runahead execution, a thread - either in hardware or software - is used to execute future instructions in an effort to prefetch useful data which is otherwise difficult to prefetch.

These works have the potential to effectively remove H2P branches and irregular memory access cache misses. These do not, however, target specifically the load-branch dependency explored in this study. When traversing a data structure, for example, runahead execution is not able to execute far enough in the future due to pointer chasing. In addition, if any type of work is done on each node, this type of loop cannot be transformed as proposed in CFD.

List of References

[1] F. Dahlgren and P. Stenstrom, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 4, pp. 385–398, 1996.

[2] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[3] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 469–480.

[4] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.

[5] M. Karlsson, F. Dahlgren, and P. Stenstrom, "A prefetching technique for irregular accesses to linked data structures," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE, 2000, pp. 206–217.

[6] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 285–297.

[7] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998, pp. 115–126.

[8] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 117–127.

[9] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

[10] J. J. Yi and D. J. Lilja, "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Transactions on computers*, vol. 55, no. 3, pp. 268–280, 2006.

[11] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[12] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, p. 84–97, May 2003. [Online]. Available: https://doi.org/10.1145/871656.859629

[13] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 329–340.

[14] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 269–280.

[15] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 2003, pp. 129–140.

CHAPTER 3

Motivation

3.1 Overlap Latency

As mentioned in the introduction, H2P branches and irregular data accesses constitute a large amount of branch mispredictions and cache misses in state-of-the-art CPUs. These also tend to be interrelated, adding additional complexity to consider when attempting to improve performance. In this section, the concept of overlap latency is introduced, which demonstrates the problems that arise due to this relationship.

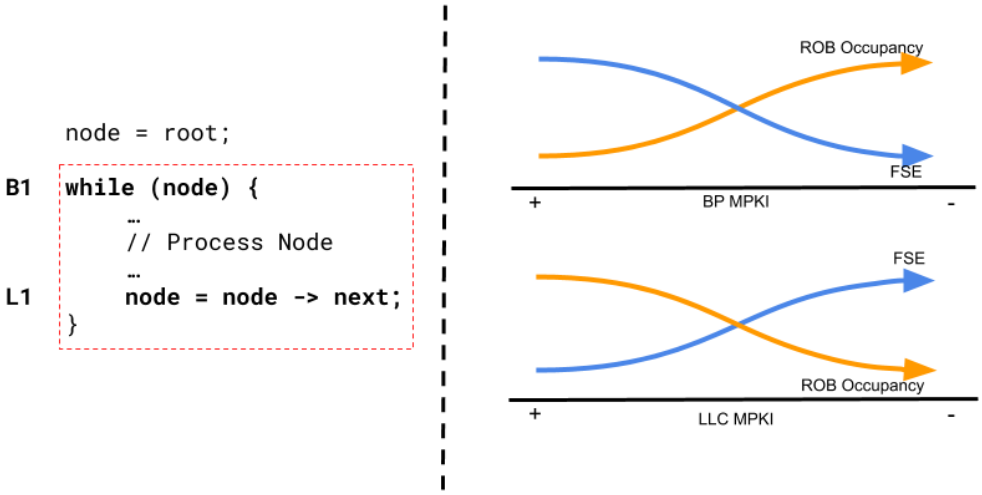


Figure 1: Basic concept of overlap latency. Frequency of Squash Events (FSE) increases as cache MPKI decreases. ROB Occupancy increases as branch MPKI decreases.

An example of overlap latency is illustrated in figure 1. In this example, a while loop is iterated over many times throughout execution of the program. The line labeled B1 frequently results in a branch misprediction while the line labeled L1 produces a significant amount of cache misses. If work is done to improve data cache prefetching so that line L1 no longer produces many cache

misses, the processing of instructions will be much faster since instructions in the reorder buffer (ROB) no longer need to wait for long memory accesses. While faster instruction processing will lead to improved performance, the presence of line B1 will limit the amount performance can improve. This is because the faster instruction processing will also result in more frequent branch mispredictions from line B1. So while more instructions can now be fetched, many of these instructions will later be flushed from the pipeline as a result of line B1. Conversely, if branch prediction is improved so that B1 no longer results in branch mispredictions, more useful instructions will be fetched. While this will make more efficient use of the pipeline since all processed instructions will later be committed, the long memory accesses caused by cache misses from line L1 will lead to the ROB holding more instructions. As the ROB fills, the pipeline will have to delay execution of further instructions which limits the potential performance improvements made possible by better branch prediction.

This load-branch dependency created by lines L1 and B1 lead to what is referred to in this paper as overlap latency. Overlap latency refers to the parallel latencies caused by cache misses (i.e. L1) and branch misprediction penalties (i.e. B1). Since both of these sources of latency occur frequently as the loop is executed, removing only one source will not provide significant performance improvement due to the presence of the other source. As will be shown in the results section, applications with significant overlap latency can be severely limited in potential performance gain unless both the load and branch are handled in tandem.

3.2 Motivating Example

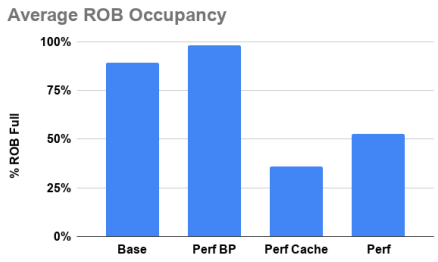
As an example of this type of behavior, the benchmark MST from the Olden benchmark suite will be used. This benchmark computes the minimum spanning tree of a graph. A bottleneck within this benchmark is the while loop shown

in figure 3. This loop performs a hash table key lookup. In this case, a cache miss occurs frequently when retrieving $ent = hash- > array[j]$ from memory. In addition, the check to see if ent is valid (not NULL) results in frequent branch misprediction. As can be seen, this represents a load-branch dependency in which the load is caused by an irregular data access (linked list) and the branch is a H2P branch since its result is dependent on this irregular data access.

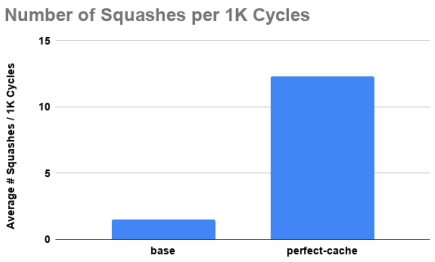
If branch prediction were to be improved, this would lead to an increased ROB occupancy since many instructions will be waiting for memory accesses. Conversely, if data cache prefetching is improved, this would lead to an under-utilization of the ROB given the increased frequency of squash events, i.e. branch mispredictions. This is shown in figure 4, where data obtained from a detailed simulation was used to plot the average ROB occupancy and the frequency of squash events for four different configurations: baseline (current state-of-the-art), perfect branch prediction (perfect-bp), perfect cache (perfect-cache), and perfect branch prediction and perfect cache (perfect). If there was no co-dependency between lines B1 and L1, one would expect contributions from perfect branch prediction would be independent to contributions from perfect cache performance. However, as can be seen from the IPC values shown in figure 4, the actual performance gain seen from the perfect configuration is much higher than this expectation. The additional speedup made possible by removing both latencies together is referred to in this work as overlap speedup.

In order to explain this overlap speedup, figure 4 shows results obtained from tracking the for loop during execution. These results were obtained by periodically taking a snapshot of the instructions within the ROB and monitoring these instructions until they were all either committed or squashed. Iterations were represented by a single predetermined instruction which is executed on each loop iteration. The

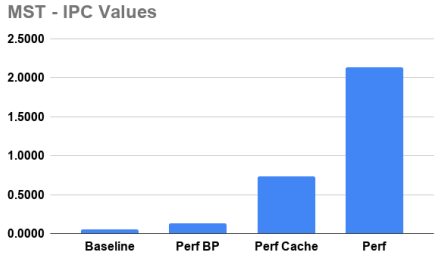
iteration commit ratio shows the average number of iterations actually committed compared to the average number of iterations found in the ROB in each snapshot. As the results indicate, the perfect-bp configuration commits every iteration found in the ROB at the cost of increased time to process the instructions due to memory latency. Perfect-cache, on the other hand, processes instructions in the ROB very quickly, however, it only commits about half of the iterations processed. The perfect configuration experiences both a high iteration commit ratio as well as fast instruction processing. Therefore, while perfect-bp improvements are limited by the presence of long latency operations and perfect-cache improvements are limited by the presence of frequent ROB flushes, the perfect configuration is able to unlock a significant amount of additional performance gain by eliminating both of these overlapping latencies.



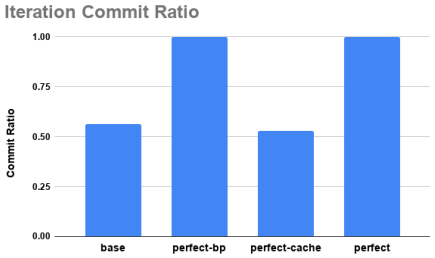
(a) ROB Occupancy



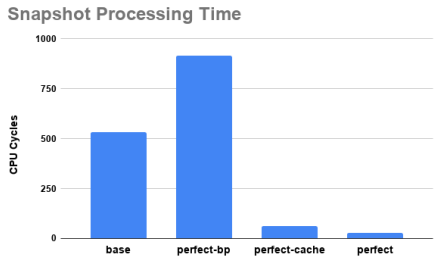
(b) Frequency of Squash Events



(c) IPC of each configuration



(d) Average iteration commit ratio



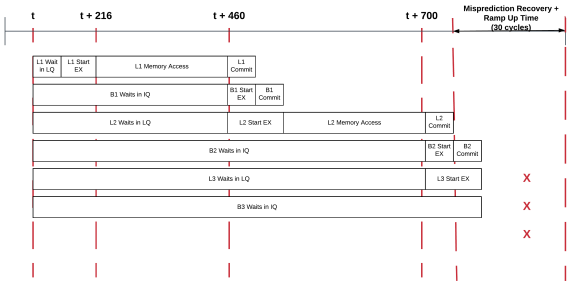
(e) Average time to process ROB snapshot

Figure 2: Effects of overlap latency in the pipeline.

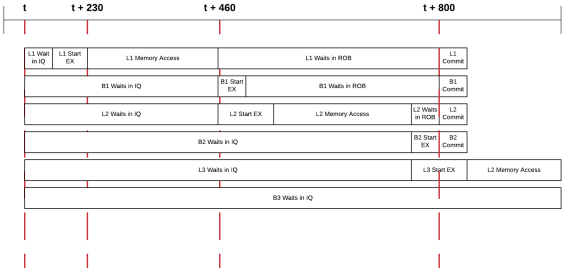
```
      void *HashLookup(unsigned int key, Hash hash)
      {
        int j;
        HashEntry ent;

        j = (hash->mapfunc)(key);
        assert(1,j>=0);
        assert(2,j<hash->size);
B1      for (ent = hash->array[j]; ent && ent->key!=key;
C1           ent=ent->next);

        if (ent) return ent->entry;
        return NULL;
      }
```

```
|     ...
|     0x401e67:    mov    0x0(%rbp),%rdx
|     0x401e6b:    cltq
|     0x401e6d:    mov    (%rdx,%rax,8),%rax
|     0x401e71:    test   %rax,%rax
|     0x401e74:    jne    0x401e89 <HashLookup+57>
|     0x401e76:    jmp    0x401e91 <HashLookup+65>
|     0x401e78:    nopl   0x0(%rax,%rax,1)
|     0x401e80:    mov    0x10(%rax),%rax
|     0x401e84:    test   %rax,%rax
|     0x401e87:    je     0x401e91 <HashLookup+65>
|C1   0x401e89:    cmp    %ebx,(%rax)
|B1   0x401e8b:    jne    0x401e80 <HashLookup+48>
|     ...
```

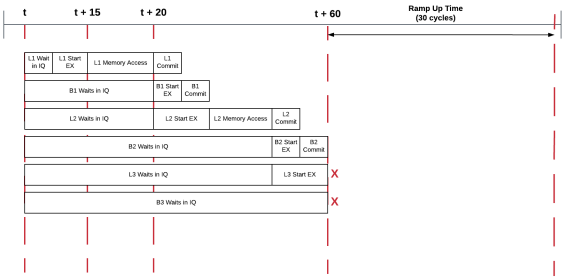
Figure 3: Code snippet of MST.



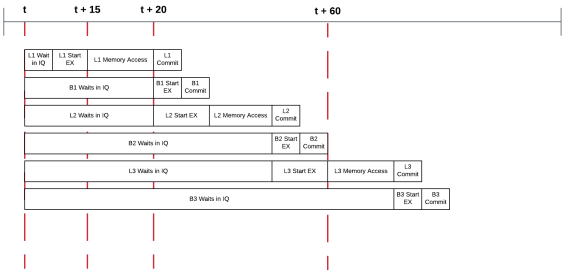
(a) Memory and branch latencies present (base).



(b) Branch latency removed, cache misses still present (perfect-bp).



(c) Cache misses removed, branch mispredictions still present (perfect-cache).



(d) Both memory and branch latencies removed (perfect).

Figure 4: Pipeline view demonstrating effects of overlap latency.

CHAPTER 4

Methodology

4.1 Workloads

For this study, it was of interest to investigate a wide range of coding styles to determine the set of algorithms, traversals, or other attributes which cause parallel latency in the pipeline. To this end, the following benchmark suites were selected:

Olden[1] is a computation-instensive benchmark suite which makes extensive use of linked data structures. Olden has been used in many studies aiming to mitigate the effect of pointer-chasing. These benchmarks were chosen since they have been well researched and they target applications with a large amount of irregular data accesses.

The Problem Based Benchmark Suite (PBBS)[2] is a C++ based benchmark suite designed to study applications which ultize an algorithm to solve some problem. For example, this benchmark suite includes benchmarks such as comparison sort and the travelling salesman problem. This suite was chosen since it represents many types of useful algorithms found in real-world applications.

The Asynchronous Access Memory Chaining (AMAC)[3] benchmark suite is an extension of a collection of C main memory hash joins that have been optimized for a specific multi-core CPU/compiler pairing (Intel Xeon x5670, gcc 4.7.2). A design goal of AMAC is to hide memory latency within pointer-intensive data structures (i.e. hash tables) via keeping state information for each lookup separate from other lookups. This benchmark suite was chosen due to its extensive use of hash table algorithms.

The Graph Algorithm Platform Benchmark Suite (GAPBS)[4] is a C++ framework whose goal is to accelerate graph processing research via offering standardized graph processing baseline implementations. GAP provides very high

performance implementations for each kernel through the use of complex and intuitive algorithms. This benchmark suite was chosen since it provides evaluation for many popular graphing algorithms and has been used in many papers.

CRONO[5] is a C multi-threaded graph analytic benchmark suite developed at the University of Connecticut, Storrs. Released roughly at the same time as GAPBS, CRONO aims to become a standard multi-threaded graph algorithm benchmark suite. CRONO benchmarks leverage C array data structures, and uses a combination of direct and indirect access patterns. Comparing and contrasting CRONO with GAPBS, both suites aim to be a standard graph based benchmark suite, with two different implementations (based in C and C++, respectively) and access patterns. For this study, the multi-threaded functionality has been set to run on a single thread.

The Standard Performance Evaluation Corporation CPU 2017 benchmark suite (SPEC CPU 2017)[6] is the de facto computer architecture research benchmark suite, consisting of a collection of computational intense performance benchmarks that stress the processor, memory hierarchy, and compilers. This benchmark suite also offers the opportunity to evaluate real-world applications, rather than algorithms which are incorpprated into applications. For this study, only the speed integer benchmarks were selected. Since the focus of this research is on single core performance, the rate benchmarks are not of interest. In addition, recent characterizations of the floating point benchmarks have shown small amounts of branch misprediction.

4.2 Simulation Configuration

To obtain detailed measurements from the collection of benchmark suites, the gem5 simulator[7] was used. All simulations were run in syscall emulation (SE) mode on an x86 ISA. The baseline configuration for the simulated CPU is shown in

Table 1: Simulated CPU Configuration

Core	Out-of-Order, 2GHz
Pipeline	256-Entry ROB, 96 LQ Entries, 64 SQ Entries
Branch Pred.	TAGE-SC-L
L1 Data Cache	64kB, 4-cycle latency, 8-way, 24 mshr entries
L2 Cache	256kB, 12-cycle latency, 8-way, 24 mshr entries
LL Cache	4MB, 32-cycle latency, 16-way, 48 mshr entries

Table 1. All workloads were compiled statically with the O2 setting and the -fno-ssa-phiopt flag. Checkpoints for each benchmark were generated in one of two ways, depending on the complexity of the benchmark. For benchmarks which represent some kernel or data structure traversal, a single checkpoint was taken immediately before the region of interest. These single checkpoints were then simulated using 5 million warmup instructions and 100 million detailed simulation instructions. For other benchmarks which cannot be accurately measured using a single checkpoint, such as SPEC CPU2017, the SMARTS methodology was utilized. Each checkpoint generated was then simulated for 5 million warmup instructions and 1 million detailed simulation instructions.

Simulations were run using four different configurations: state-of-the-art, perfect-bp, perfect-cache, and perfect. The state-of-the-art configuration used the TAGE-SC-L branch predictor[8]. Results for this configuration were obtained using four different prefetchers: Stride, SPP, BOP, and IMP. The perfect-bp configuration emulated a perfect branch predictor (direction only) while still using a state-of-the-art prefetcher. The perfect-cache configuration emulated an L1 data cache which never missed while using the TAGE-SC-L branch predictor. The perfect configuration emulated both perfect branch prediction and perfect L1 data cache. The implementation details of perfect-bp and perfect-cache can be found in the following section.

4.3 Implementations

4.3.1 Perfect Branch Prediction

In order to evaluate the potential contributions possible made by branch prediction, a method for ensuring perfectly accurate branch prediction was needed to use in simulations. In this study, branch prediction was considered to be the prediction of taken or not taken only. This means branch mispredictions are still possible in the event of a BTB miss or an incorrect BTB predictions. The frequency of these mispredictions vary depending on workload, however for a vast majority of benchmarks considered these events were very infrequent. To achieve perfect branch prediction, two modifications were made to the GEM5 simulator.

First, the correct path for a particular workload must be recorded. Therefore, a preliminary simulation is run in which the taken/not taken information for every committed instruction is recorded in an external text file in the order they are committed. Since the information recorded is just a boolean value for every control instruction, this does not require a large storage space.

Once the correct path was recorded for the workload, it was then run in the perfect branch prediction configuration. The external file is loaded into the simulator as an array. On every prediction, the correct outcome is extracted from the array and the branch predictor's decision is overridden by this value. As mentioned earlier, mispredictions due to the BTB are still possible. Because of this, squashes are still possible so the array pointer must be able to return to a previous state when this happens. This is handled using a small circular buffer the size of the ROB, which tracks the dynamic instructions which have caused the pointer to increment. When a tracked instruction is squashed, the pointer is also decremented.

4.3.2 Perfect Cache

To evaluate memory access latency effects on performance, a perfect cache was needed to emulate a L1 data cache with a 100% hit rate. Therefore, all memory accesses would incur the penalty of an L1 cache access only which is the goal of all data cache prefetchers. In order to implement this in GEM5, different timing modes available in GEM5 were taken advantage of. In order to simulate memory access times, GEM5 uses a timing mode which emulates the detailed interactions between the CPU and all levels of the memory hierarchy. GEM5 uses has a functional mode, which is traditionally used to load workload information into simulator memory at the beginning of a simulation or when a functional simulation is run which does not track memory details. The perfect cache implemented for this study utilizes the GEM5 timing mode to simulate interactions between CPU and L1 data cache. The interactions between L1 and L2 cache, however, were overridden so that the functional mode was used to retrieve the appropriate data from lower levels without incurring any penalties. This implementation essentially creates an L1 data cache the size of main memory with minimal changes to existing GEM5 code.

4.4 Metrics Used

For this study, several common metrics were used, as well as metrics developed specifically to describe behavior related to overlap latency. The following are common metrics that were collected from simulations:

- BP MPKI - branch mispredictions per kilo-instruction
- L1D MPKI - L1D cache misses per kilo-instruction
- LLC MPKI - LLC cache misses per kilo-instruction
- IPC - Instructions per Cycle

- Hot PC - Instruction which produces a relatively large number of misses or mispredictions compared to other instructions in benchmark.

4.4.1 Load-Branch MPKI

Overlap latency is a consequence of a load-branch dependency which is executed many times throughout the execution of an application. Moreover, only cache misses (i.e. access patterns which are not prefetched well in state-of-the-art mechanisms) and H2P branches contribute to overlap latency. Therefore, only benchmarks which exhibit a high number of both branch mispredictions and cache misses should be considered for this work. In order to identify such benchmarks, a metric referred to as *Load-Branch MPKI* was used.

$$LoadBranchMPKI = \frac{MPKI_{BP} * MPKI_{CACHE}}{MPKI_{BP} + MPKI_{CACHE}} \tag{1}$$

$$MPKI_{CACHE} = MPKI_{L1D} + MPKI_{LLC} \tag{2}$$

This equation makes use of the observation that a branch miss and a cache miss can be viewed as parallel processes in this work. The reason for this is that both branch and cache misses are required to have overlap latency, and therefore if one or the other is not present that is no opportunity for overlap latency.

4.4.2 Expected Speedup

The aim of this work is to prove that there is additional performance gain made possible by removing both branch mispredictions and cache misses together. This additional gain cannot be achieved by either branch prediction or cache prefetching since the other source of latency is still present in either case. Therefore, the actual speedup observed via simulation must be compared to the speedup that would be expected based on the performance of perfect-bp and perfect-cache. Since the assumption is that there is no relationship between performance gain from branch

prediction and cache, the expected speedup is defined as:

$$ExpectedSpeedup = Speedup_{perfect-bp} * Speedup_{perfect-cache} \quad (3)$$

since these speedups should be independent of one another.

4.4.3 Overlap Speedup

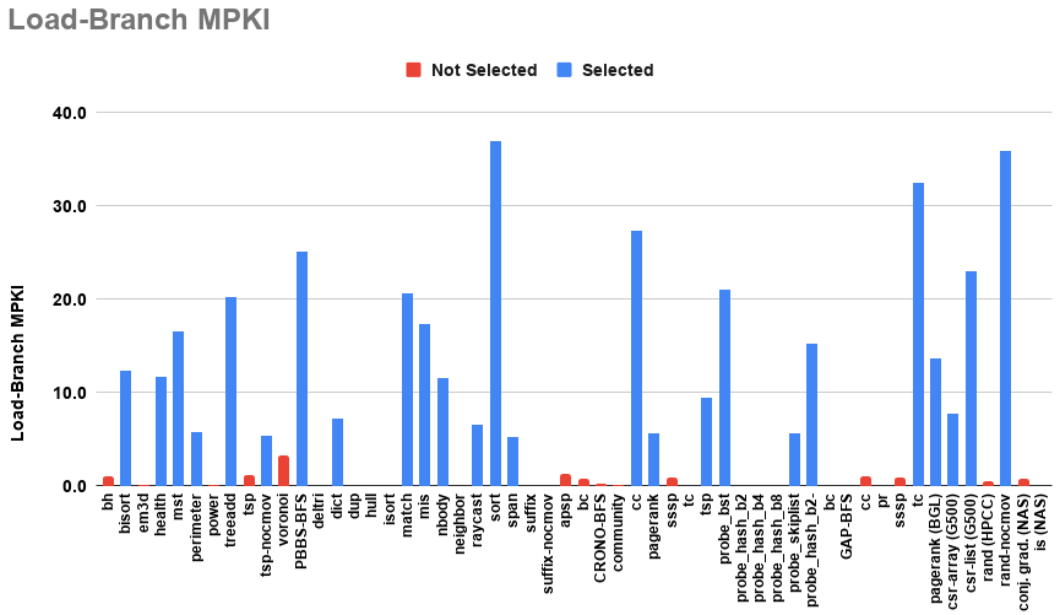
Overlap speedup is the metric used to describe the additional speedup a benchmark can possibly achieve by removing the latencies caused by both the load and branch together. This is speedup that is not possible by improving the latency of just one source. Overlap Speedup is defined as:

$$OverlapSpeedup = \frac{Speedup_{perfect}}{ExpectedSpeedup} \quad (4)$$

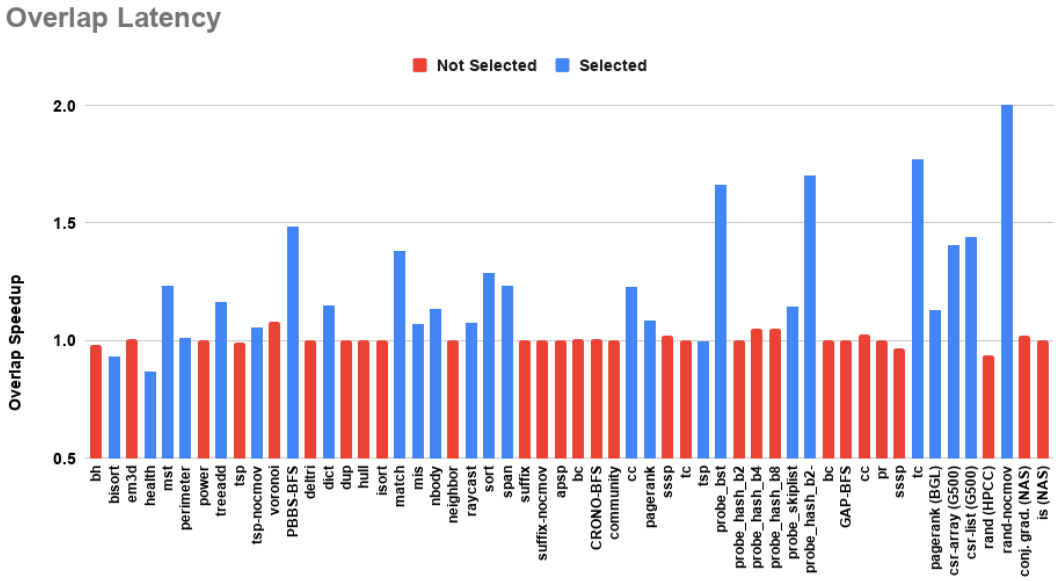
Since expected speedup refers to the speedup expected by removing all branch mispredictions and cache misses based on the simulation results of perfect-cache and perfect-bp, overlap speedup is able to capture any additional speedup gained by completely removing the load-branch dependency.

The Load-Branch MPKI as well as overlap speedup for each benchmark simulated is shown in figure 5. All benchmarks with a high load-branch MPKI exhibit a significant amount of overlap speedup, and all benchmarks with a low load-branch MPKI exhibit negligible overlap speedup.

Two benchmarks, bisort and health, have significant overlap speedup however it is actually signigicantly lower than expected. The reason for this is that the perfect-cache configuration actually helps branch prediction, which violates the assumtion that perfect-bp and perfect-cache are independent. In these cases, the benchmarks still mispredict on the same instructions, however, they have to have far fewer predictions which lowers the opportunity for mispredictions. This is due to improved cache performance allowing branch mispredictions to be caught much faster.



(a) Load-Branch MPKI



(b) Overlap Latency

Figure 5: Load-Branch MPKI values for all benchmarks simulated. Comparing this to the overlap latency values, it can be seen that no benchmarks with a low Load-Branch MPKI exhibit significant overlap latency.

List of References

[1] M. C. Carlisle, "Olden: parallelizing programs with dynamic data structures on distributed-memory machines," Ph.D. dissertation, Princeton University, 1996.

[2] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 68–70.

[3] O. Kocberber, B. Falsafi, and B. Grot, "Asynchronous memory access chaining," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 252–263, 2015.

[4] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[5] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 44–55.

[6] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[8] A. Seznec, "Tage-sc-l branch predictors again," 2016.

CHAPTER 5

Analysis

In this chapter, benchmarks selected based on Load-Branch MPKI will be analyzed. First, a software analysis is provided demonstrating algorithms which are vulnerable to overlap latency. Then, architectural strain caused by latency overlap is shown to explain why the load-branch dependency created in software limits the performance of a benchmark.

5.1 Software-Level Analysis

After careful analysis of a large range of benchmarks, those which showed a large amount of overlap latency were able to be grouped into categories. In this section, these categories will be discussed along with one or more examples which provide concrete evidence of how the overlap is created in software.

Table 2: Workload Categorization

Algorithm Type	Benchmarks
Neighboring Node Access	BFS, CC, Match, MIS, Pagerank, TC
Hash Table Lookups	BST, Hash, Skiplist, MST, Dict, RandAcc
Linked Data Structure Travsal	Treeadd, TSP, Health, Perimeter
Data Dependent Modifications	Sort, CSR-List, Bisort

1. **Neighboring Node Access** This class of algorithm typically traverses over all nodes (e.g. vertices) in the data structure. In each iteration, these algorithms will access or modify other nodes in the structure. Since these other nodes are not necessarily sequential, this creates an irregular access pattern that is hard to predict. Therefore, a frequent cache miss is created from

loading the other nodes within each iteration. In addition, any type of decision which must be made based on these other nodes will result in a H2P branch, causing frequent branch mispredictions. This combination results in a load-branch dependency, leading to significant overlap latency if this loop is iterated over many times throughout execution.

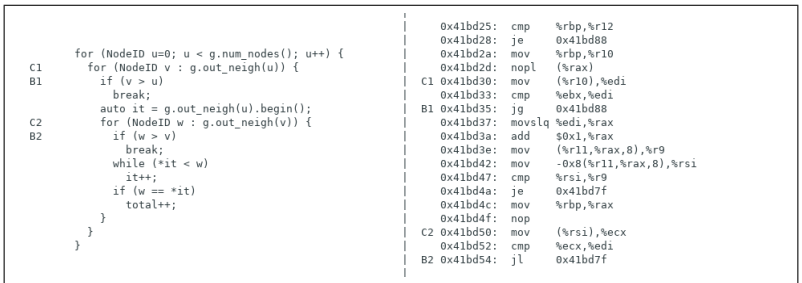


Figure 6: Code snippet of TC.

An example of this class of algorithm is TC from the GAP benchmark suite. This algorithm implements a triangle counting algorithm which determines the number of triangles found in the graph provided as input. In order to make this calculation, TC implements a function named OrderedCount. This function traverses over an ordered set of vertices and attempts to determine if each vertex forms a triangle. As can be seen from figure 13, this is implemented using a nested for loop. Both the outer and inner for loop create a load-branch dependency by accessing and traversing over all neighbors of a certain vertex. Two examples of this load-branch dependency are labelled in figure 13 (B1C1, B2C2). From a 100 million instruction detailed simulation, it was found that B1 makes up 44% of all branch mispredictions and B2 27% of all mispredictions. In addition, C1 causes 39% of all cache misses and C2 causes 28%. This leads to an overlap speedup of 1.7 (i.e. an additional 70% speedup is possible due to overlap latency).

2. **Hash Table Lookups** Hash table lookups in which a key must be found

matching some other key. Two types of implementations were seen in the benchmarks studied for this paper. The first implementation uses a linked list of arrays to store entries. A hashing function determines the index the matching key would be found, and then a traversal of the linked list is performed checking each array for a possible match. In this case, a load-branch dependency is created by traversing the linked list until a key is found. Pointer chasing is repsponible for the irregular memory access, and the unpredictability associated with the length of each linked list as well as the value of each entry creates the H2P branch. An example of this is seen in MST which is shown as the motivating example.

The other implementation, which is used by the AMAC benchmarks, utilizes a fifo (linked list) to hold some number of keys. These fifos are stored in some data structure which must then be traversed to find matching keys. In this implementation, the load-branch is produced similar to the other implementation.

```
      while (i < rel->num_tuples){                                          |   ...
        k = k >= SIZE ? 0: k;                                               |   0x402969:   shl    $0x4,%rcx
        if(fifo[k].ptr) {                                                   |   0x40296d:   add    (%r14),%rcx
          comp = fifo[k].ptr->tuple.key-fifo[k].key;                        |C1 0x402970:   mov    %r10,(%rcx)
B1        if (comp == 0){                                                   |   0x402973:   mov    %rsi,0x8(%rcx)
C1          outR->tuples[fifo[k].payload].key = fifo[k].ptr->tuple.key;     |   0x402977:   add    $0x1,%edx
            ...                                                             |   0x40297a:   movslq %edx,%rsi
            i++;                                                            |   0x40297d:   mov    %rsi,%rcx
            fifo[k].ptr = bt->nodes;                                        |   0x402980:   shl    $0x4,%rcx
            ...                                                             |   0x402984:   add    (%r12),%rcx
B1        } else if (comp > 0){                                             |C1 0x402988:   mov    (%rcx),%rcx
            ...                                                             |   0x40298b:   mov    %rsi,0x10(%rax)
            fifo[k].ptr=fifo[k].ptr->right;                                 |   0x40298f:   mov    %rcx,0x8(%rax)
          } else {                                                          |   0x402993:   mov    0x0(%r13),%rcx
            ...                                                             |   0x402997:   mov    %rcx,(%rax)
            fifo[k].ptr=fifo[k].ptr->left;                                  |   0x40299a:   cmp    %r8d,%edx
          }                                                                 |   0x40299d:   jae    0x4029c8 <probe_bst+312>
        } else {                                                            |   0x40299f:   cmp    %edi,%ebx
          i++;                                                              |B1 0x4029a1:   jg     0x402938 <probe_bst+168>
C1        fifo[k].key = rel->tuples[i].key;                                 |   0x4029a3:   mov    $0x1,%edi
          fifo[k].ptr = bt->nodes;                                          |   0x4029a8:   xor    %eax,%eax
          ...                                                               |   0x4029aa:   jmp    0x40293e <probe_bst+174>
        }                                                                   |   0x4029ac:   nopl   0x0(%rax)
        k++;                                                                |B1 0x4029b0:   jle    0x4029f0 <probe_bst+352>
      }                                                                     |   ...
                                                                            |
```

Figure 7: Code snippet of BST.

As an example, BST will be used. This benchmark probes an unbalanced BST searching for a set of of keys. Each node of the BST contains a fifo (linked list) containing some number of keys. Whenever a key is found, the BST pointer is reset to the root and the search for the next key begins. The

unpredictability of when a key will be found (data dependent) as well as the differing sizes of each node's fifo create a H2P branch at B1. In addition, irregular data access patterns are created by accesses different pointer fields on every iteration. In this example, there are several instructions which could create a load-branch dependency, however the if-else control flows guarentees exactly 1 load-branch dependency is created on every iteration. In figure 7, these instructions are identified as B1 and C1. The two B1 instructions (combined here since they are cascaded) make up 95% of all branch mispredictions in the simulation. The two C1 instructions account for 99% of all cache misses. The large number of load-branch dependencies caused by this loop lead to an overlap speedup of 1.66.

3. **Linked Data Structure Traversal** Linked data structure traversals in which each node of some structure - such as a linked list or binary tree - is accessed according to some order. In this case, pointer chasing due to access the next node in the structure results in an irregular memory access while the composition (i.e. is the node a leaf node or not) of the linked data structure produces a H2P branch. Since in many cases there is a small amount of work to be done on each individual node, these traversals usually lead to very small loops which are iterated over many times. Since a majority of these loops is the actually load-branch dependency, these traversals can produce significant overlap latency.

   The benchmark treeadd will be used as an example. This benchmark traverses a binary tree, accumulating a running sum of the values of node. The tree is traversed postorder. The conditional checking whether or not the current node is valid (not NULL) results in a large number of branch mispredictions. In addition, the pointer chasing resulting from traversing the

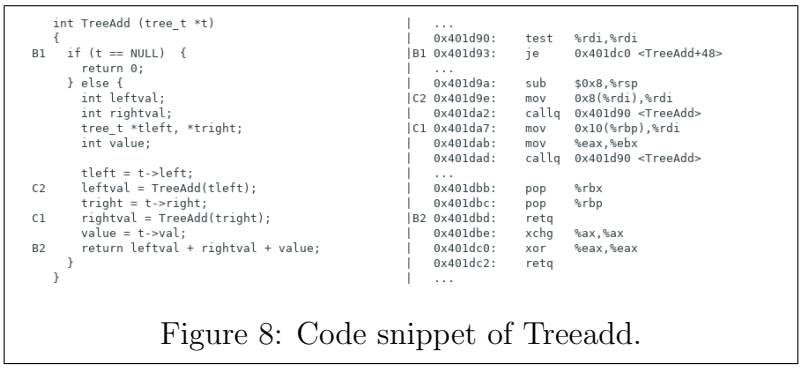


Figure 8: Code snippet of Treeadd.

children of each node results in a large number of cache misses. This loop results in a load-branch dependency identified in figure 8. The recursive call to retrieve the sum of the current node's children create the loads while the base case results in the branch misprediction. The instruction B1 is responsible for 95% of all branch mispredictions, while C1 and C2 combine to create 99% of all cache misses. This leads to an overlap speedup of 1.16.

4. **Data Dependent Modifications** Data dependent traversals such as sorting algorithms can lead to overlap latency. In these cases a H2P branch is produced given the dependence of the result of the branch on the input data. Cache misses occur due to the large amount of data being accessed and modified at once. As an example, the benchmark Comparison Sort will be used. This benchmark uses the Standard Template Library (STL) function sort to sort a random set of float data. The sort function utilizes an algorithm called IntroSort which partition the array based on some pivot. The function used to create this partition is shown in figure 9. Within this while loop values of the array are accessed from front to back as well as back to front. In addition, the front and back pointers are incremented and decremented within the loop. This creates a very difficult memory access pattern to predict, leading to cache misses. In addition, the number of iterations traversing front to back (as well as back to front) to highly dependent on

the input data. This creates a H2P branch. In this case two load-branch dependencies are created, both of which are executed on every iteration of the outer loop. The instructions B1 and B2 result in 77% of branch mispredictions and the instructions C1 and C2 account for 86% of all cache misses. This results in an overlap speedup of 1.29.

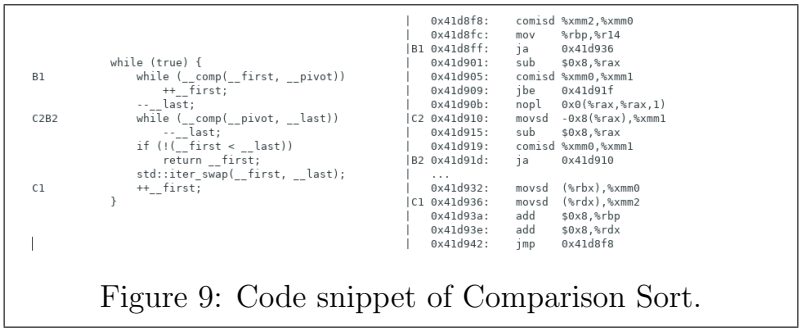


Figure 9: Code snippet of Comparison Sort.

5.2 Hardware-Level Analysis

This section will explore the effects overlap latency can have on the CPU resources. First, the consequences of improved branch prediction on benchmarks with significant overlap latency will be explained. Next, the consequences of improved cache is shown. Finally, the effects of the combination is shown which will highlight the strain put on the CPU by improved branch predcition or cache performance in isolation in these benchmarks.

Perfect Branch Prediction Consequences

The perfect-bp simulation results provide the upper bound performance for each benchmark when branch prediction direction is always correct[1]. While improved branch prediction is generally a good thing in any benchmark, this can

[1]As mentioned in the methodology section, mispredictions are still possible even if the correct direction (taken or not taken) is used. These mispredictions come from indirect jumps and jumps in which the target of a taken branch can vary during runtime. These events, at least in the benchmarks discussed here, are very rare however, and perfect branch prediction direction results in around 99.9% accuracy in almost all benchmarks simulated.

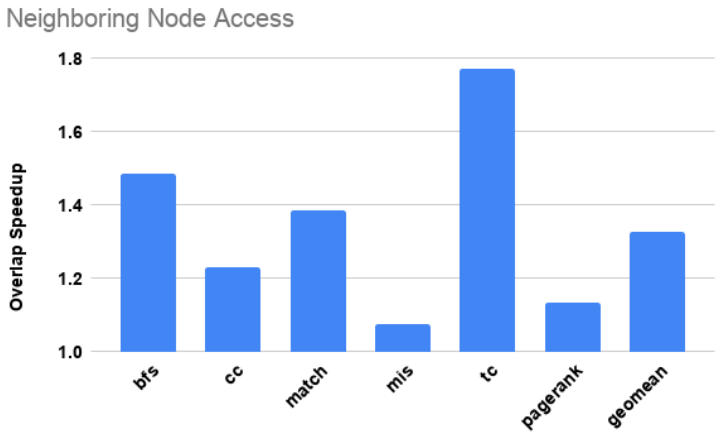


Figure 10: Possible overlap speedup found in neighboring node access benchmarks.

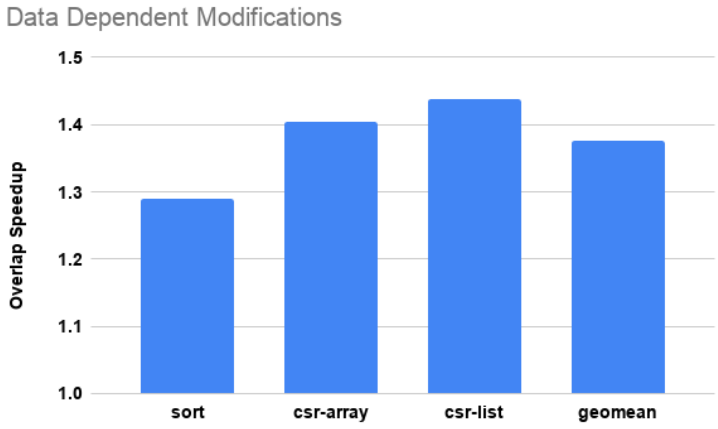


Figure 11: Possible overlap speedup found in data dependent modification benchmarks.

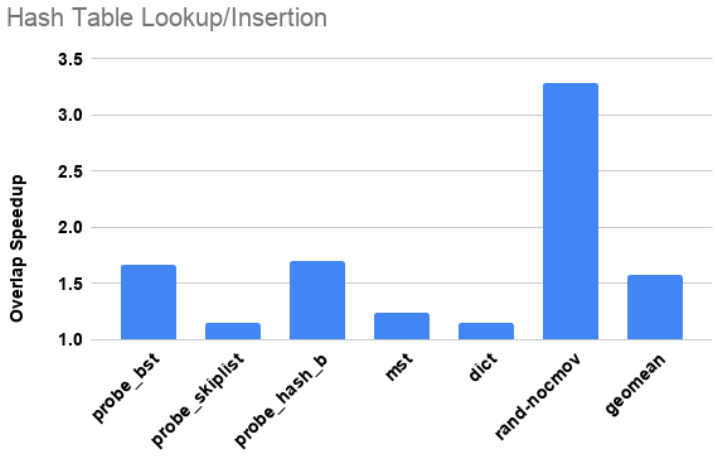


Figure 12: Possible overlap speedup found in hash table lookup/insertion benchmarks.

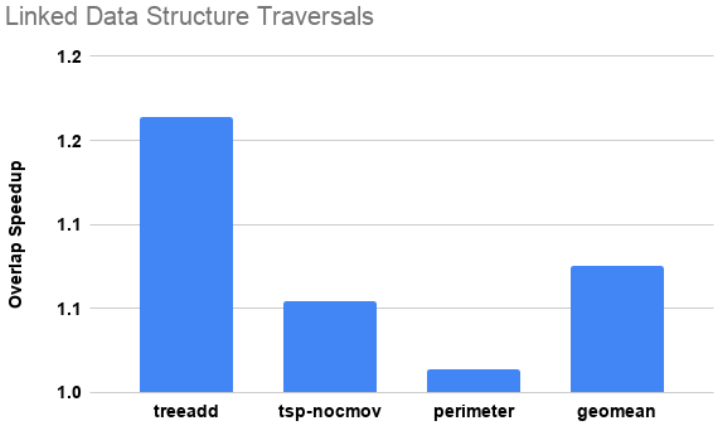


Figure 13: Possible overlap speedup found in linked data structure traversal benchmarks.

add strain to the CPU especially in benchmark's with significant overlap latency. This strain is due to the increase in useful instruction (i.e. instructions which will not be squashed) which must be processed. This is quantified in figure 14, which graphs the average ROB occupancy of each benchmark for base and perfect-bp configurations. The increase in ROB occupancy from perfect branch prediction is expected since this will drastically reduce the number of instructions squashed in the ROB. However, due to overlap latency, this increase can be even greater since this will lead to more instructions in the ROB waiting for memory accesses to return. This over-utilization of the ROB can prohibit further instructions from being executed, reducing the performance gain possible.

Perfect Cache Consequences

The perfect-cache simulation results show the upper bound limit of performance gain possible by improving cache performance. In benchmarks with overlap latency, improved cache performance alone is not enough to acheive the best possible performance. This reason for this is the presence of the branch misprediction in the load-branch dependency which creates overlap latency. While a perfect cache will allow for faster processing of instructions, H2P branches will remain to limit performance. In the case of a benchmark with overlap latency, improved cache means more frequent mispredictions and therefore more frequent squashes. The increased frequency of branch mispredictions, which is shown in figure 14, will result in an underutilized ROB. This will limit performance gain possible as there will be cycles in which no instructions will be present in the ROB.

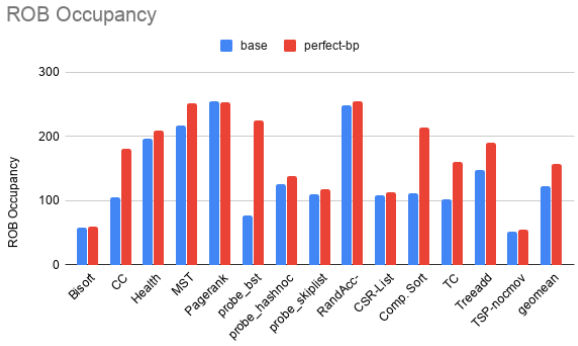
Idle Commit Cycles

The combination of of the previous two sections leads to this metric. In perfect-bp, there is still are relatively large number of cycles in which no instruc-

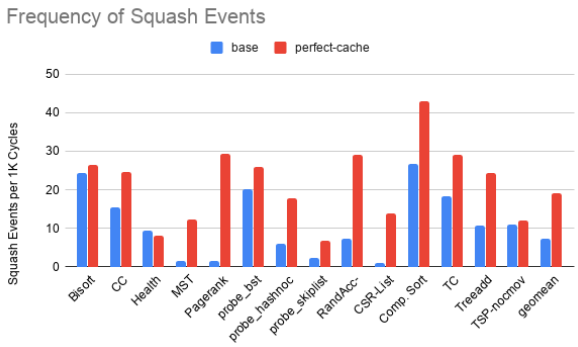
tions are committed, in many cases the head instruction is waiting for memory access. In perfect-cache, there is still significant cycles in which no instructions are committed due to frequent ROB flushes resulting in no instructions ready to commit. However, in the perfect case instructions can be processed faster due to improved memory and the ROB is utilized well so that instructions are usually ready to be committed. This observation was quantified using the ratio of cycles in which no instructions are committed to total cycles simulated. This metric does not track the number of instructions committed at each cycle, which would result in IPC. Rather, it tracks the latency of the head instruction (or lack of head instruction) in each configuration. This is useful because the size of the ROB is irrelevent. The results of this metric for each benchmark is shown in figure 14. From the geometric mean of these benchmarks, it can be seen in both perfect-bp and perfect-cache over half of all cycles result in no instructions being committed on average. In the perfect case, however, the average is reduced all the way only about a quarter of the cycles resulting in no committed instructions.

Iteration Throughput

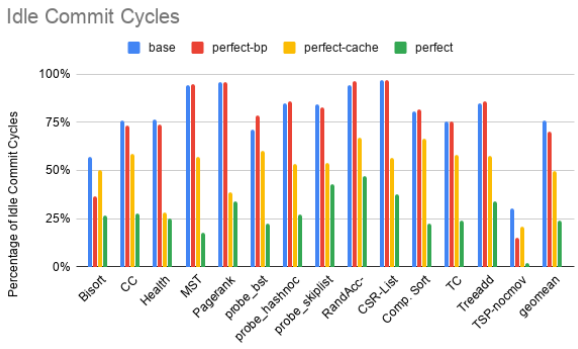
In order to provide examples of the aforementioned architectural strain placed on the CPU in the cases of both improved branch prediction and improved cache, this section will examine the throughput of each benchmark analyzed in the software analysis section. This was done by identifying the loop containing overlap latency (shown in the software analysis section) and tracking the performance of the loop during simulation. The performance was measured using three metrics: snapshot iterations (number of iterations within the ROB at some point in time), commit iterations (number of snapshot iterations which are eventually committed), and snapshot time (number of cycles needed to process instructions in the ROB when a snapshot is taken). The results of this analysis for the benchmarks discussed



(a) ROB Occupancy



(b) Frequency of Squash Events



(c) Percent of idle commit cycles compared to total cycles simulated.

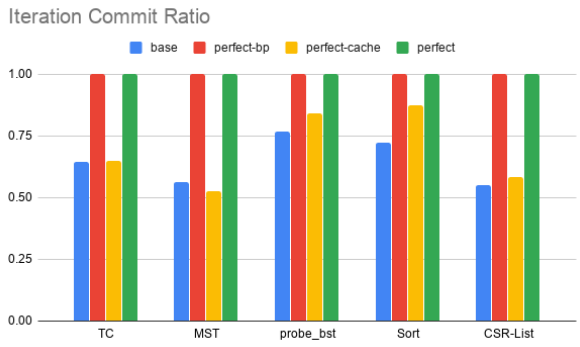
Figure 14: Effects of overlap latency in the pipeline. As branch prediction is improved, the load latency which remains leads to an increase in instructions waiting to commit. As the ROB fills, the number of instructions which can be fetched decreases. As cache misses are reduced, the branch latency which remains leads to more frequent ROB flushes. This leads to the ROB being under-utilized such that there may be no instructions in the ROB ready to commit. The average idle commit cycles tracks the latency of the head instruction from the ROB. This combines latency from perfect-bp and perfect-cache to show the added benefit in the perfect configuration.

in the software analysis section can be seen in figure 15. As expected, perfect-bp leads to 100% of snapshot iterations being committed while perfect-cache has a similar commit ratio to base. This increased number of useful instructions, however, requires more CPU cycles to commit due to memory latency. On the other hand, perfect-cache is able to process each snapshot very quickly although many processed instructions are not actually needed since they are later squashed.

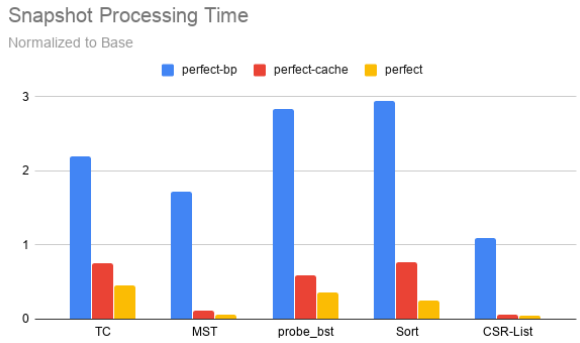
Effect of the ROB Size

The ROB serves many purposes, one of its main purposes is to hide long memory access latency by allowing other instructions to execute while the CPU waits for the data to return from memory. By increasing the size of the ROB, the CPU is afforded more opportunity to hide these memory latencies therefore reducing the effect of cache misses on overall performance (i.e. IPC). Because of this, the ROB plays a significant role in the impact overlap latency has on a benchmark's performance. While the base configuration for this study utilizes an ROB around the size of typical modern CPUs, it is also of interest to examine how larger ROBs will effect overlap latency in the future.

To do this, simulations were run on all selected benchmarks while varying the size of the ROB from 256 entries up to 1024 entries. This also involved increasing the number of physical registers, instruction queue entries, and load/store queue entries by the same ratio as the ROB. In the perfect-bp configuration, a large majority of benchmarks saw either small changes or significant IPC improvements as the ROB size increased. This is to be expected, since reducing the impact of cache misses will lead to more reliance on branch prediction when overlap latency is present. Also as expected, the perfect-cache configuration saw very minimal changes in almost all benchmarks. Finally, the perfect configuration saw no change in IPC which is also expected. Since in the perfect configuration the ROB does



(a) Ratio of Iterations Tracked to Iterations Committed

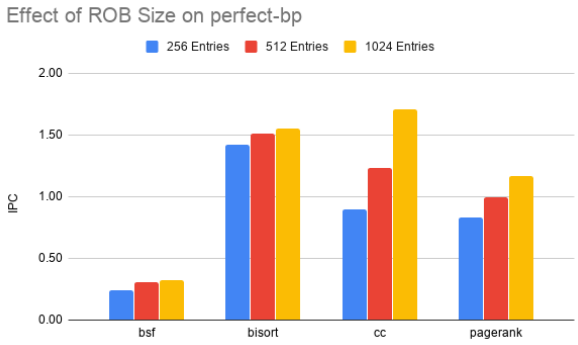


(b) CPU cycles required to process a snapshot, normalized to baseline.

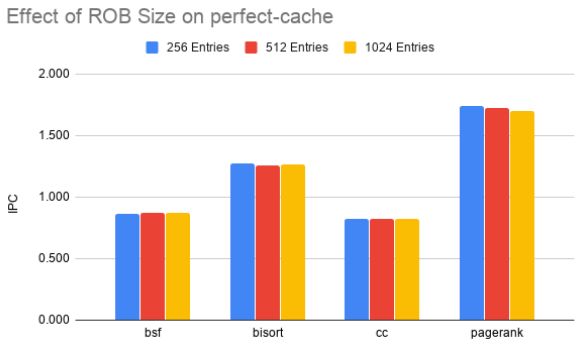
Figure 15: Iteration throughput for several representative benchmarks. As seen in the figure, perfect-bp increases useful iterations processed at the cost of longer processing time per iteration. Conversely, perfect-cache reduces the amount of time to process an iteration while wasting CPU resources by processing iterations that will later be squashed.

not have to hide any long latency operations or suffer and branch misprediction penalties, a large ROB is not needed or utilized.

The combination of improved IPC from branch prediction along with no change in the perfect configuration leads to most benchmarks experiencing less overlap latency as the ROB increases. Although benchmarks saw a reduced amount of overlap latency as the ROB increased, almost all benchmarks which had some amount of overlap latency with 256 ROB entries still experienced overlap latencies with an ROB as large as 1024 entries. These results are shown in figure 16.



(a) IPC as ROB size increases for perfect-bp.



(b) IPC as ROB size increases for perfect-cache.

Figure 16: Effect of ROB size on branch prediction and cache.

CHAPTER 6

Results

In the previous section, the causes of overlap latency as well as the effects it can have on the processor were analyzed. In this chapter, the impact this has on the actual performance of these benchmarks will be demonstrated. To show this, the upper bound IPC values for all four configurations obtained via detailed simulation will be shown. A deeper analysis of these results will then provide better insight into just how limiting latency overlap has the potential to be on certain applications.

6.1 Upper Limit IPC

In this section, the impact overlap latency can have on a benchmark's performance when present. This was investigated by simulating each benchmark using four configurations: base (TAGE-SC-L + SPP) which represents current state-of-the-art implementations, perfect-bp which emulates a branch prediction *direction* while maintaining SPP prefetching, perfect-cache which emulates a perfect L1 *data* cache while maintaining TAGE-SC-L branch prediction, and perfect which emulates both perfect branch prediction direction and perfect L1 data cache. Figure 17 shows the IPC values of all four configurations of the selected benchmarks.

As can be seen by the IPC values, these benchmarks see a much higher performance improvement in the perfect configuration compared to the other three. While this may be expected given perfect contains both perfect-bp and perfect-cache benefits, this paper argues that there are additional, and less obvious, benefits to improving both branch prediction and cache together which unlock much greater potential for performance improvement in benchmarks with overlap latency. As an example, the benchmark MST (discussed in the motivation section)

Table 3: Geometric mean of additional speedup compared to expected speedup found in each class of benchmark.

Category	AVG. Additional Speedup
Neighboring Node Access	30.56%
Hash Table Lookups	23.88%
Linked Data Structure Traversal	7.30%
Data Dependent Modifications	34.50%

is not able to obtain an IPC over 1.0 with either perfect-bp or perfect-cache. However, when both are combined the upper bound on the IPC raises dramatically to over 2.0.

Figure 19 offers a slightly different perspective of how to view the simulation results. This figure shows the impact overlap latency has on a benchmark by plotting the additional speedup seen in the perfect configuration compared to what was expected based on the perfect-bp and perfect-cache configurations. For example, the benchmark TC sees about 75% more speedup in the perfect configuration that its calculated expected speedup (5.75 in the perfect compared to 3.25 expected speedup). Of the benchmarks selected based on Load-Branch MPKI, up to 229% additional speedup was found (this was the RandAcc benchmark) with an average of 16% across all benchmarks. Table 3 shows the average additional speedup found by category (as defined in the previous chapter).

While it can be seen from these results that the full potential for performance improvement cannot be achieved without removing both sources of latency together in these benchmarks, this work also aims to show that as one source of latency is removed, the removal of the other grows in importance. This can be seen in table 6. This table shows the speedup made possible by branch prediction and by cache. It shows this speedup for the case when the other source of latency is present as well as when the other source of latency is removed as well. As can

Table 4: Branch prediction results for all selected benchmarks.

Benchmark	BP Accuracy	BP MPKI
bisort	88.16%	21.508
health	93.44%	16.490
mst	94.76%	17.432
perimeter	95.89%	12.393
treeadd	88.25%	33.577
tsp-nocmov	95.52%	8.080
bfs	84.84%	29.540
dict	94.34%	8.340
match	88.29%	22.838
mis	93.50%	18.841
nbody	91.62%	12.745
raycast	94.79%	7.688
sort	75.77%	62.232
span	97.01%	5.539
cc	86.21%	34.463
pagerank	93.95%	6.819
tsp	90.75%	14.459
probe_bst	86.39%	35.958
probe_skiplist	96.58%	6.606
probe_hash_b2-nocmov	92.16%	20.542
tc	71.32%	67.434
pagerank (BGL)	91.25%	14.824
csr-array (G500)	97.01%	8.033
csr-list (G500)	90.63%	24.323
rand-nocmov (HPCC)	74.66%	44.113

Table 5: Cache data for all selected benchmarks.

Benchmark	L1D Miss Rate	L1D MPKI	LLC Miss Rate	LLC MPKI
bisort	0.060	28.228	0.057	0.332
health	0.106	34.310	0.996	6.127
mst	0.416	211.231	0.783	95.676
perimeter	0.018	7.761	0.912	2.843
treeadd	0.069	42.092	0.998	8.401
tsp-nocmov	0.045	16.072	0.129	0.339
bfs	0.224	117.919	0.886	48.227
dict	0.193	29.257	0.894	21.026
match	0.266	140.698	0.914	70.777
mis	0.375	156.856	0.645	68.913
nbody	0.287	106.417	0.986	10.539
raycast	0.112	41.171	0.658	4.743
sort	0.179	85.985	0.980	4.955
span	0.296	86.538	0.915	31.876
cc	0.137	128.260	0.195	3.052
pagerank	0.079	26.429	0.723	5.248
tsp	0.007	18.745	0.999	8.682
probe_bst	0.064	24.421	0.615	26.203
probe_skiplist	0.085	28.348	0.604	12.601
probe_hash_b2-nocmov	0.054	21.188	0.990	37.241
tc	0.125	53.663	0.842	8.716
pagerank (BGL)	0.485	145.684	0.415	31.607
csr-array (G500)	0.455	192.209	0.531	53.041
csr-list (G500)	0.704	250.229	0.786	158.696
rand-nocmov (HPCC)	0.269	101.408	0.989	92.306

be seen, branch prediction is more effective (produces higher speedup) when cache misses are removed for almost all benchmarks simualted. Simiarly, cache produces higher speedups when branch mispredictions are removed. It is for this reason that these benchmarks achieve a significant amount of additional speedup in the perfect configuration compared to expected speedup.

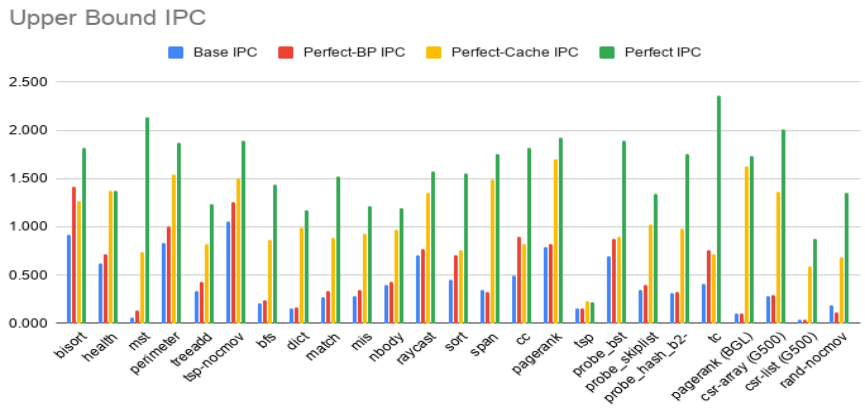


Figure 17: Upper bound limit of IPC for each configuration simulated. Benchmarks such as BST and TC see a large larger potential in the perfect configuration compared to perfect-bp and perfect-cache.

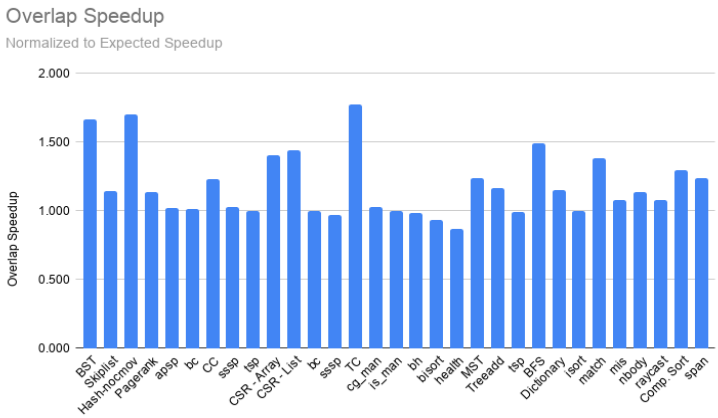


Figure 18: Overlap speedup. This is the extra speedup obtained due to removing both load and branch latency compared to the expected speedup based on perfect-cache and perfect-bp results.

Table 6: Speedup Results

Benchmark	BP Speedup	Cache Speedup	BP Speedup (perfect-cache)	Cache Speedup (perfect-bp)
bisort	1.546	1.378	1.441	1.285
health	1.155	2.203	1.001	1.911
mst	2.347	13.039	2.899	16.109
perimeter	1.197	1.855	1.213	1.880
treeadd	1.291	2.466	1.502	2.871
tsp-nocmov	1.192	1.421	1.257	1.499
bfs	1.117	4.044	1.659	6.009
dict	1.029	6.156	1.183	7.078
match	1.237	3.218	1.711	4.452
mis	1.214	3.300	1.303	3.541
nbody	1.085	2.453	1.231	2.784
raycast	1.085	1.904	1.168	2.049
sort	1.588	1.694	2.048	2.185
span	0.954	4.303	1.178	5.316
cc	1.801	1.644	2.216	2.023
pagerank	1.045	2.152	1.134	2.335
tsp	0.994	1.471	0.993	1.470
probe_bst	1.265	1.299	2.105	2.161
probe_skiplist	1.146	2.922	1.311	3.344
probe_hash_b2-nocmov	1.052	3.170	1.790	5.393
tc	1.846	1.759	3.272	3.119
pagerank (BGL)	0.943	15.111	1.067	17.097
csr-array (G500)	1.049	4.868	1.473	6.840
csr-list (G500)	1.034	14.947	1.488	21.507
rand-nocmov (HPCC)	0.601	3.613	1.977	11.880

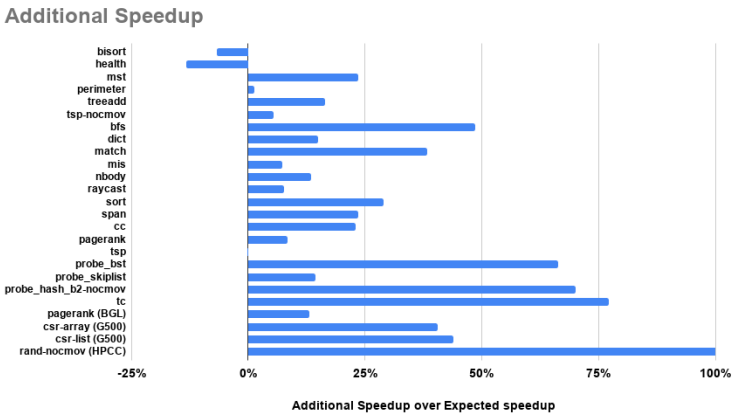


Figure 19: Additonal speedup seen in the perfect configuration compared to expected speedup calculated from perfect-bp and perfect-cache results.

CHAPTER 7

Discussion

7.1 SPEC CPU2017

7.1.1 Framework Limitations

While the simulation framework used in this study was able to accurately measure the overlap latency found within simple benchmarks which evaluate a single graph traversal or algorithm, there are some limitations when this method is applied to more complex benchmarks such as SPEC CPU2017.

Benchmark Complexity

Since SPEC CPU2017 benchmarks are meant to represent real-world applications, as well as the complexity associated with real-world implementations, they cannot be accurately evaluated by simulating just one section of each application. Unlike previous benchmarks explored, there is no single region of interest in which to focus the detailed simulations. To handle this, as mentioned in the background section, the SMARTS methodology[1] was utilized to take checkpoints throughout the lifecycle of each benchmark. These checkpoints were taken using the Lapdary tool developed by a group of researchers at the University of Michigan. This tool allowed the checkpoints to be generated using GDB running on native hardware, as opposed to running the benchmark in the simulator. By generating checkpoints on native hardware, a significant amount of time was saved (i.e. several hours compared to several weeks). In order to customize this tool for this particular work, wrapper code was written to automate the process of determining the correct interval at which to take checkpoints and to store the generated checkpoints in the correct location. Based on prior work done[2], it was determined that approximately 100 checkpoints per benchmark would be sufficient to obtain accurate

simulation results for SPEC 2017. In order to reconcile these multi-checkpoint benchmarks with the other single-checkpoint benchmarks, an average of statistics collected from all checkpoints was used to represent the performance of the benchmark. This is able to give a good representation of, for example, the unpredictibility of branches found in a benchmark (i.e. branch MPKI). The averaging, however, tends to hide other attributes of the benchmark, such as areas where latency overlap limit performance.

Therefore, in order to conduct detailed analysis on the SPEC benchmarks, checkpoints of interest were selected from each benchmark and each of these checkpoints were treated as individual simulations. This will be discussed further in the analysis section.

Indirect Branches

Another limitation of the framework used in this study is the increased use of indirect branches and calls found within the SPEC CPU2017 benchmarks. Indirect jumps can cause branch mispredictions even when the direction is correctly predicted if the target of the taken branch is predicted wrong. While other benchmarks do not make frequent use of indirect jumps, SPEC benchmarks have a significant amount of these. Since perfect branch prediction was defined as perfect direction only, this limits the upper bound estimate for perfect-bp.

7.1.2 Impact of Overlap Latency

Although it is important to point out the limations of the methodology used, useful analysis was still conducted on the SPEC benchmarks. In this section, the overall impact of overlap latency will be explored and then a detailed analysis of select areas of the MCF benchmark will be provided showing the presence of overlap latency.

Figure 20 shows the overlap latency indicator function values for the SPEC benchmarks. Applying the same threshold to these benchmarks as was applied to the previous benchmarks, one can see that only MCF has significant opportunity for overlap speedup. This observation was confirmed by simulation, as the overlap speedups for each benchmark are shown in figure 22. In the next section, a detailed analysis of MCF will be given.

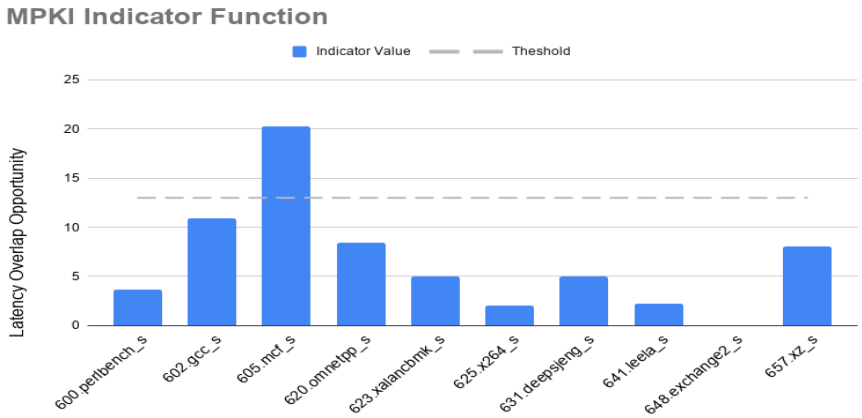


Figure 20: SPEC Load-Branch MPKI Values. In many SPEC benchmarks, there is either a large amount of performance to be gained from only one source of latency (i.e. low Load-Branch MPKI).

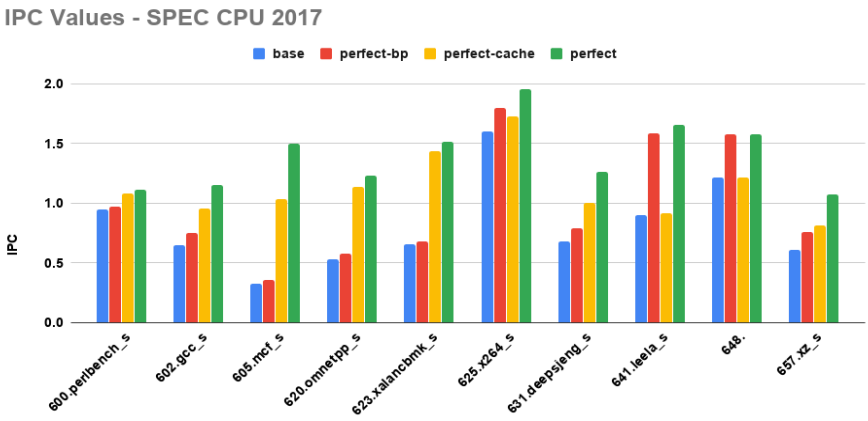


Figure 21: SPEC IPC values obtained using all four configurations. As expected from Load-Branch MPKI values, most benchmarks see an increased IPC from either perfect-bp or perfect-cache but not both.

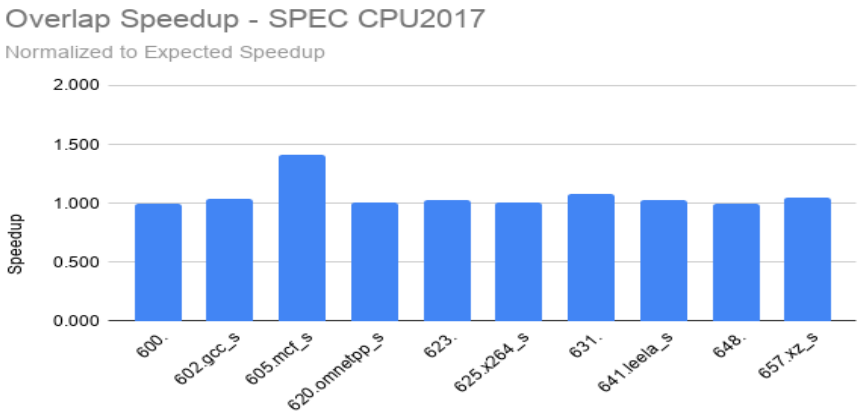


Figure 22: SPEC Overlap Latency, compares well to indicator functions.

MCF

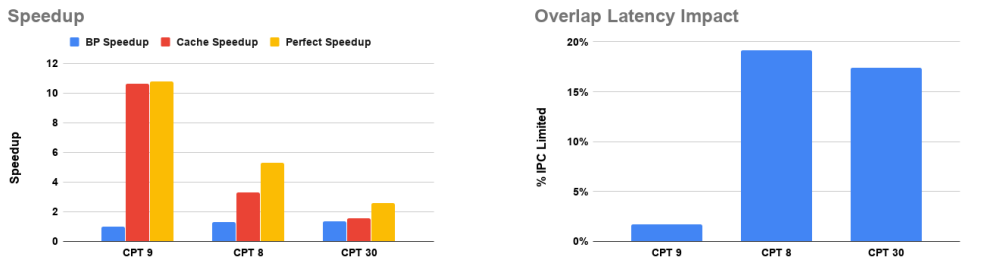
In order to examine the presence and impact of overlap latency found in MCF, results from three separate checkpoint simulations will be shown. These checkpoints represent different stages of the execution lifecycle of MCF. While it is true that different parts of a program should be weighted based on how often the part is executed, that information is provided by the averaging of all checkpoints since if one part of a benchmark is executed often more than one checkpoint will execute that part. In this section, the interest lies in how different parts of MCF operate rather than overall performance impact.

The first checkpoint that will be examined, referred to as CPT 9, does not contain overlap latency. While both branch mispredictions and cache misses occur frequently in this checkpoint, they occur at different stages of execution thus avoiding overlap. The effect of this can be seen in figure 23 where performance improvement is almost completely dominated by cache behavior. For this checkpoint, speedup results are very close to expected, shown in figure 23, meaning there is little additional benefit to improving cache and branch prediction together.

In contrast to CPT 9, two other checkpoints were chosen which do contain

overlap latency. The cause of the load-branch dependencies are from different execution stages, as shown in figure 24. Because of the load-branch dependency in these checkpoints, neither perfect-bp nor perfect-cache are able to achieve speedups approaching that of perfect. This impact is shown again in figure 23.

As these examples demonstrate, latency overlap can still have an impact on long, complex benchmarks however this impact is not as dramatic overall as that seen in previous examples. This is to be expected, however, as previous benchmarks are meant to expose a particular algorithm to find bottlenecks while SPEC benchmarks are meant to evaluate real world applications.



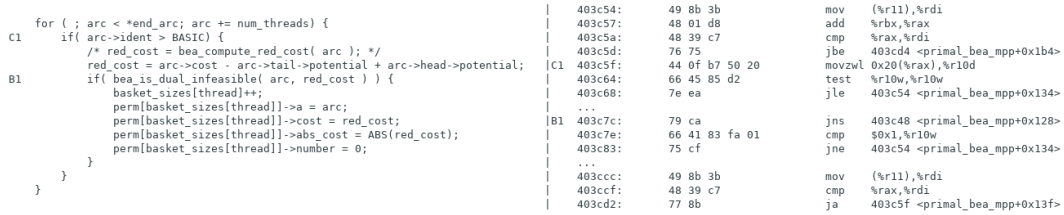
(a) MCF - ROB Occupancy (b) MCF - Frequency of Squash Events

Figure 23: Effects of overlap latency in the pipeline for MCF.

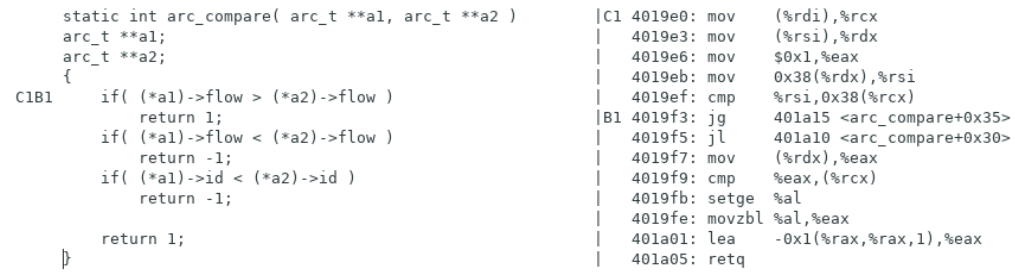
7.2 Impact of cmov

An important consideration when analyzing overlap latency within a benchmark is the use of cmov instructions. Modern compilers use cmov instructions when deemed more efficient than relying on branch prediction. If a compiler does choose to use cmov rather than a branch and load, it can drasticly reduce the number of branch predictions made. While in many cases the compiler does a good job of deciding when and where to place cmov instructions, future improvements to branch prediction could impact this decision. Therefore, the use of cmov instructions was monitored during this study.

To demonstrate the effect of cmov, three benchmarks simulated in this study



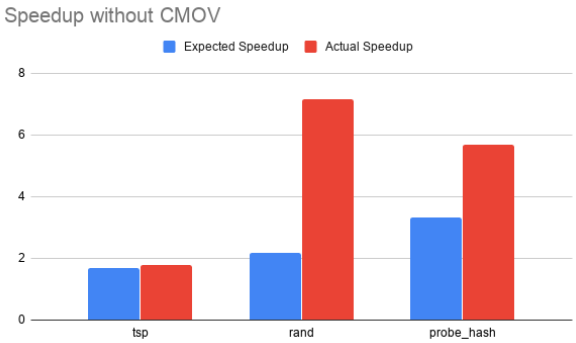
(a) CPT 30



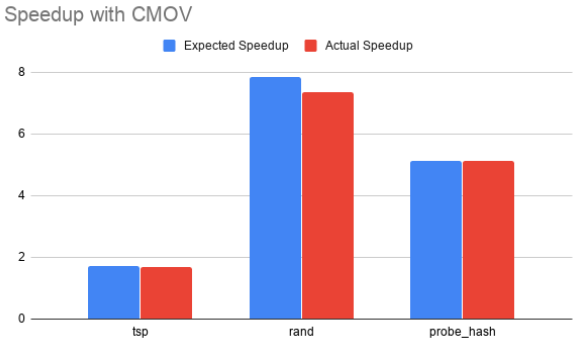
(b) CPT 8

Figure 24: MCF source code which results in a load-branch dependency. These executed in different stages of execution in MCF, leading to an increased overlap latency overall.

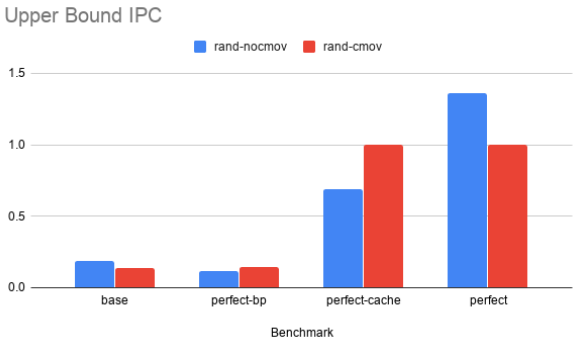
will be used. In figure 25, the potential speedup of these benchmarks is shown when compiled with cmov and instructions as well as without cmov. The cmov instructions were disabled using the flags *-fno-ssa-phiot -fno-if-conversion -fno-if-conversion2 -fno-tree-loop-if-convert -fno-tree-loop-if-convert-stores*. As can be seen, the use of cmov severly limits the amount of overlap latency in a benchmark. This can be an advantage, especially in current state-of-the-art CPUs, however as branch prediction and cache performance is improved, the use of cmov does not allow for as much IPC improvement. As an example of this, the potential IPC values for the RandAcc benchmark is shown. In the base configuration, the use of cmov results in a higher IPC. In addition, improvements to cache significantly increase IPC when cmov is used compared to without cmov. However, when both branch mispredictions and cache misses are reduced, the use of cmov limits the potential IPC by more than 25%.



(a) Speedup with cmov instructions.



(b) Speedup without cmov instructions.



(c) Upper bound IPC for the RandAcc benchmark both with an dwithout cmov.

Figure 25: Effect of cmov on specific benchmarks. While cmov can be benficial in current CPU architectures, it limits potential performance improvements made possible by removing overlap latency.

List of References

[1] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, p. 84–97, May 2003. [Online]. Available: https://doi.org/10.1145/871656.859629

[2] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586.

CHAPTER 8

Conclusion

In this study, it was shown that a load-branch dependency formed by H2P branches and irregular data accesses can significantly impact the potential performance gain for some types of benchmarks. First an indicator function which relates branch MPKI and cache MPKI to overlap latency opportunity was provided. This function was then used to narrow down the set of benchmarks that were analyzed further. These benchmarks of interest were then simulated to show the upper bound speedup made possible by perfect branch prediction alone, perfect L1 cache alone, and perfect branch prediction and perfect L1 cache together. In all selected benchmarks, there existed some amount of additional performance gain unlocked by removing both sources of latency in tendem. The additional speedup was termed overlap speedup. The cause of overlap speedup in different categories of benchmarks was then shown from a software perspective. Finally, the effects of the load-branch dependency on the CPU was examined in an attempt to explain the additional speedup. This was shown by explaining the increased importance of branch prediction as cache improves (and vice versa) in benchmarks with this load-branch dependency.

This work provides a foundation for future research into practical implementations that attempt to reduce both sources of latency together. By providing upper bound limits on performance and showing the additional performance made possible, this work shows provided motivation for more active research into this area. In addition, by providing categories of algorithms which are vulnerable to overlap latency, possible starting points for this new research has been given.

BIBLIOGRAPHY

"Championship branch prediction (cbp-5)," 2016. [Online]. Available: https://www.jilp.org/cbp2016/

"The 3rd data prefetching championship," 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/

Ahmad, M., Hijaz, F., Shi, Q., and Khan, O., "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 44–55.

Ayers, G., Litz, H., Kozyrakis, C., and Ranganathan, P., "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 513–526.

Bakhshalipour, M., Tabaeiaghdaei, S., Lotfi-Kamran, P., and Sarbazi-Azad, H., "Evaluation of hardware data prefetchers on server processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, 2019.

Beamer, S., Asanović, K., and Patterson, D., "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

Braun, P. and Litz, H., "Understanding memory access patterns for prefetching," in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, 2019.

Bucek, J., Lange, K.-D., and v. Kistowski, J., "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

Carlisle, M. C., "Olden: parallelizing programs with dynamic data structures on distributed-memory machines," Ph.D. dissertation, Princeton University, 1996.

Casper, J. and Olukotun, K., "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 151–160.

Dahlgren, F. and Stenstrom, P., "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 4, pp. 385–398, 1996.

Hennessy, J. L. and Patterson, D. A., *Computer architecture: a quantitative approach*. Elsevier, 2011.

Hill, M. D. and Marty, M. R., "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

Jiménez, D. A. and Lin, C., "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

Karlsson, M., Dahlgren, F., and Stenstrom, P., "A prefetching technique for irregular accesses to linked data structures," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE, 2000, pp. 206–217.

Kim, J., Pugsley, S. H., Gratz, P. V., Reddy, A. N., Wilkerson, C., and Chishti, Z., "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

Kocberber, O., Falsafi, B., and Grot, B., "Asynchronous memory access chaining," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 252–263, 2015.

Lin, C.-K. and Tarsa, S. J., "Branch prediction is not a solved problem: Measurements, opportunities, and future directions," *arXiv preprint arXiv:1906.08170*, 2019.

Michaud, P., "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 469–480.

Mittal, S., "A survey of techniques for dynamic branch prediction," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 1, p. e4666, 2019.

Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. N., "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 2003, pp. 129–140.

Parkhurst, J., Darringer, J., and Grundmann, B., "From single core to multi-core: preparing for a new exponential," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 67–72.

Peled, L., Mannor, S., Weiser, U., and Etsion, Y., "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 285–297.

Perelman, E., Hamerly, G., Van Biesbrouck, M., Sherwood, T., and Calder, B., "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

Purser, Z., Sundaramoorthy, K., and Rotenberg, E., "A study of slipstream processors," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 269–280.

Roth, A., Moshovos, A., and Sohi, G. S., "Dependence based prefetching for linked data structures," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998, pp. 115–126.

Seznec, A., "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 117–127.

Seznec, A., "Tage-sc-l branch predictors again," 2016.

Sheikh, R., Tuck, J., and Rotenberg, E., "Control-flow decoupling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 329–340.

Shun, J., Blelloch, G. E., Fineman, J. T., Gibbons, P. B., Kyrola, A., Simhadri, H. V., and Tangwongsan, K., "Brief announcement: the problem based benchmark suite," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 68–70.

Weisse, O., Neal, I., Loughlin, K., Wenisch, T. F., and Kasikci, B., "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586.

Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C., "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, p. 84–97, May 2003. [Online]. Available: https://doi.org/10.1145/871656.859629

Yi, J. J. and Lilja, D. J., "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Transactions on computers*, vol. 55, no. 3, pp. 268–280, 2006.

Yu, X., Hughes, C. J., Satish, N., and Devadas, S., "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.